Parallel Bounded Analysis in Code with Rich Invariants by Refinement of Field Bounds

Nicolás Rosner,
FCEyN - UBA
Buenos Aires, Argentina

Juan Galeotti,
Saarland University
Saarbrücken, Germany

Santiago Bermúdez,
Guido Marucci Blas,
Santiago Perez De Rosso,
Lucas Pizzagalli,
Luciano Zemín,
Marcelo F. Frias
ITBA
Buenos Aires, Argentina

ABSTRACT

In this article we present a novel technique for automated parallel bug-finding based on the sequential analysis tool TACO. TACO is a tool based on SAT-solving for efficient bug-finding in Java code with rich class invariants. It prunes the SAT-solver's search space by introducing precise symmetry-breaking predicates and bounding the relational semantics of Java class fields. The bounds computed by TACO generally include a substantial amount of nondeterminism; its reduction allows us to split the original analysis into disjoint subproblems. We discuss the soundness and completeness of the decomposition. Furthermore, we present experimental results showing that MUCHO-TACO, our tool which implements this technique, yields significant speed-ups over TACO on commodity cluster hardware.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.1 [**Software Engineering**]: Specifications; D.2.4 [**Software Engineering**]: Program verification—Class invariants, programming by contract, formal methods.

General Terms

Verification, Languages

Keywords

Static analysis, SAT-based code analysis, Alloy, DynAlloy.

1. INTRODUCTION

Finding bugs is a key activity in software development. Even well-engineered systems are likely to contain faults. Many can be detected by testing the system, but since testing only exercises a finite, usually small number of executions, many faults can go undetected.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2159-4/13/07...$15.00
http://dx.doi.org/10.1145/2483760.2483770

Bounded Exhaustive Analysis (BEA) techniques and tools complement traditional testing. BEA approaches consist of testing a system using every possible input among those that satisfy certain imposed bounds.

Various kinds of bounds and combinations thereof can be considered. Kiasan [9], a tool for test input generation, bounds the maximum length of reference chains in the memory heap (called the k-bound). Korat [5], a tool aimed at generating non-isomorphic test inputs, bounds the number of object references per class. JForge [10], Miniatur [3] and TACO [12, 13], bug-finding tools based on SAT-solving, bound the number of object references as well as the number of loop iterations that the system under test may perform.

A tool capable of handling sufficiently large bounds would render testing obsolete. Although we are not there yet, current BEA tools can, for instance, achieve optimal mutant-killing in complex container classes [12].

TACO (Translation of Annotated COde) is a BEA tool that pushed the barriers of the field by allowing roughly twice the size of the data domains that state-of-the-art tools like JForge can handle – a significant improvement considering that analysis times generally grow exponentially as bounds are increased.

Classes that are constrained by rich class invariants often give rise to deceptively simple-looking methods. Such methods frequently prove to be harder than expected to get right because the code must preserve the complex constraints induced by the invariants. Typical examples of such classes are pointer-based generic data structures like the ones studied in this article.

Analysis at the system level boils down to analysis at the method level through the use of modular analysis techniques as presented, for instance, in [10]. The real bottleneck, then, is analysis at the intraprocedural level. Assuming adequately modularized systems, complexity rarely remains proportional to the number of lines of code. Instead, it seems to reflect something inherently complex about the code itself. We claim that this kind of complexity is usually captured by the richness of the class invariants and method contracts. Consequently, the techniques and optimizations we present are primarily aimed at improving analyzability of methods with nontrivial contracts within classes with nontrivial invariants.

Building upon our previous work on TACO, we introduce a novel technique for its parallelization and present experimental results showing significant speed-ups. The MUCHO-TACO tool allows users to leverage existing idle hardware for fast verification of correct code whose sequential analysis with TACO would potentially require hundreds of hours. Even more extreme speed-ups are commonplace when analyzing faulty code.

To the best of our knowledge, no other tools for scope-bounded exhaustive analysis are able to handle comparable scope magnitudes for equivalent problems. We elaborate on this in Section 5.

Unlike many techniques either implemented as one-off prototypes, guarded by strict licensing or for some other reason unobtainable for experimental comparison, MUCHO-TACO is available for unrestricted download [31].

The article is organized as follows. In Section 2 we describe the TACO technique. In Section 3 we present the MUCHO-TACO technique as well as 3 optimizations. In Section 4 we present experiments showing that MUCHO-TACO can achieve significant speed-ups during analysis. In Section 5 we discuss related work. Finally, in Section 6 we present our conclusions and some proposals for further work.

2. TACO

TACO is a state-of-the-art BEA tool for finding faults in Java programs. In [12] we showed that, for classes with rich class invariants, TACO outperforms the bug-finding abilities of JForge [10], Kiasan [9], ESC/Java2 [6], Java PathFinder [28] and Jahob [4]. This section is an overview of TACO; see [12, 13] for a detailed presentation.

2.1 Symmetry Breaking + Tight Bounds

User input to TACO consists of Java code annotated with JML [11] requires/ensures contracts and possibly with class invariants. The TACO tool produces a propositional formula, and uses a SAT-solver to search for a valuation that satisfies it. If one is found, a Java execution trace violating the contract is extracted from it.

Although a very similar approach is followed by tools like JForge, at least two important differences are worth noting. First, during the translation process TACO adds automatically generated symmetry-breaking predicates that canonicalize the Java memory heap representation. This alone already yields significant improvements in analysis time. We provide more details in Section 2.3. Second, using these predicates TACO pre-computes tight upper bounds for the relational semantics of Java fields, which allow for the elimination of numerous primary variables during translation to a propositional formula, resulting in considerable additional savings in analysis time. We provide more details about this in Section 2.4.

2.2 Brief Introduction to Alloy and Kodkod

TACO uses Alloy [16] and its back-end Kodkod [26] as intermediate languages during the translation of annotated code to a propositional formula. In this section we present an overview of the relevant features.

Alloy specifications are called *models*. From a Java class for singly-linked lists, TACO produces an Alloy model containing the Alloy signature hierarchy shown in Fig. 1.

Signatures (identified by the keyword `sig` in Alloy) denote sets of objects. Signatures whose declaration is preceded by

```
sig Object {}
one sig null {}
sig List extends Object {
   head : one (Node + null) }
sig Node extends Object {
   next : one (Node + null) }
```

Figure 1: Type hierarchy for singly-linked lists in Alloy.

the keyword `one` (as is the case with `null` in Fig. 1) denote singleton sets. Signatures are akin to Java classes, and may contain fields, which denote binary relations. For example, according to Fig. 1,

$$\texttt{head} \subseteq \texttt{List} \times \texttt{(Node + null)},\ \texttt{next} \subseteq \texttt{Node} \times \texttt{(Node + null)}.$$

The keyword `one` preceding the codomain definitions forces the binary relations to be total functions. Thus,

$\texttt{head : List} \to \texttt{(Node + null)}$,
$\texttt{next : Node} \to \texttt{(Node + null)}$.

Signature extension, denoted by the `extends` keyword, constrains objects from the extending signature to be part of the extended signature.

Axioms are added in Alloy using the `fact` construct. For instance, the following fact constrains all lists to be acyclic. (The Alloy operators `*` and `^` stand for reflexive-transitive closure and for transitive closure, respectively.)

```
fact Acyclic { all l : List, n : Node |
     n in List.head.*next implies n !in n.^next }
```

Alloy models may include *assertions*, which are properties to be checked by the Alloy Analyzer [16]. The following (false) assertion claims that all lists are nonempty:

```
assert NoEmptyLists {all l:List | l.head != null}
```

Before checking an assertion, the user needs to provide signature size bounds (called the *scope* of the analysis in Alloy jargon). For instance, we may write

```
check NoEmptyLists for 5
```

to check whether the property holds over all possible configurations where up to 5 lists and nodes are allowed.

Kodkod is used by the Alloy Analyzer during the translation of an Alloy model to a propositional formula. One of its distinguishing features is support for partial instances: for each Alloy signature field f, Kodkod also accepts two relations L_f (the *lower bound* for field f) and U_f (the *upper bound* for field f). The presence of nontrivial lower and/or upper bounds constrains Alloy models to satisfy the restriction $L_f \subseteq f \subseteq U_f$. We will make use of this inclusion in Section 2.4, when explaining how TACO tight bounds are computed and exploited.

2.3 Symmetry Breaking in Java Heaps

It is shown in [12, 13] that adding appropriate symmetry-breaking predicates can improve analysis times by several orders of magnitude. In this section we present a brief introduction to the symmetry-breaking predicates employed by our technique.

Let us consider a memory heap containing the singly-linked list depicted in Fig. 2(a). Solid arrows link nodes to nodes referenced through field *next*, whereas dotted ones link

nodes to the values that they store. In Fig. 2(b) we show the same heap, except for the fact that labels (i.e. memory locations) have been permuted via $\{N_0 \mapsto N_4, N_1 \mapsto N_3, N_2 \mapsto N_2, N_3 \mapsto N_1, N_4 \mapsto N_0\}$. The nodes of list (b) are located at different places in memory than those of (a). Other than that, both should be considered essentially the same list, since they contain the same elements in the same order. This becomes evident once we compare Fig. 2(a) with Fig. 2(c). Such permutations over domains are called *symmetries*, and it is well known [15] that removing them generally improves the performance of SAT-solvers.

If the code under analysis with TACO contains a bug, TACO retrieves an execution trace exposing the failure. As part of such a trace, TACO provides an initial state. Therefore, the SAT-solver must be able to find this initial state (as well as the rest of the trace).

As explained in Section 2.2, Alloy performs analyses within given scopes. Let us assume a scope of 5 for the `Node` domain; hence, all initial states considered by the SAT-solver will involve at most 5 `Node`. Among the states that will need to be considered by the solver are those illustrated by Figs. 2(a) and (b). But, as we have already argued, these represent essentially the same list. In [12] we introduced symmetry-breaking axioms that remove symmetries from the memory heap representation in Alloy. These predicates complement the lower-level symmetry-breaking axioms already included by the Alloy Analyzer [15], and remove all symmetries from the Alloy models generated by TACO from Java classes. For instance, of the lists shown in Fig. 2 and all other similar isomorphisms, only (a) is considered by TACO.

The predicates are introduced by instrumenting the intermediate Alloy model. As part of the instrumentation, constants are introduced in the model. For the singly-linked lists example, assuming a scope of 5 `Node`:

```
one sig N0, N1, N2, N3, N4 extends Node {}
```

These constants will be used to compute tight bounds in the manner described in the following section.

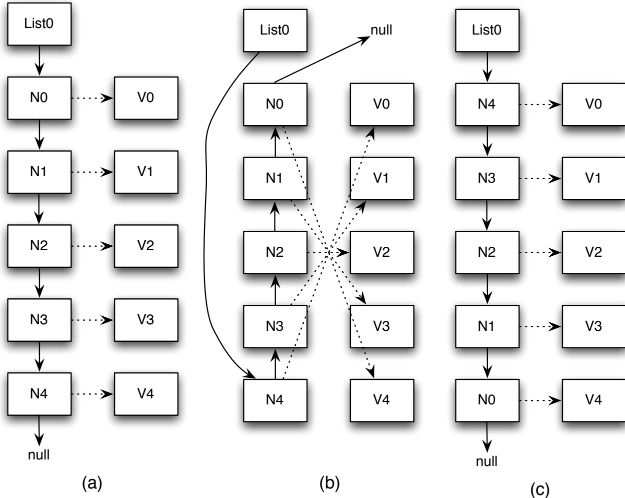

Figure 2: Isomorphic lists: (a) Original list. (b) A permutation over the nodes. (c) A redrawing of list (b).

2.4 Computing Tight Bounds

In the context of BEA tools and techniques, the term *bound* tends to acquire different (and often conflicting) meanings. Since TACO uses Alloy as an intermediate language, Alloy scopes, which *bound* the size of each data domain as seen in the previous section, become just as essential for TACO parametrization. Users of the TACO tool need to decide on appropriate scope values prior to analysis. Another relevant bound in TACO is the *loop unrolling bound* – a parameter that limits the maximum number of loop iterations allowed in the code under analysis. TACO users are expected to choose this value beforehand, according to their needs.

By *TACO bounds* or *tight bounds*, however, we mean neither of the above. TACO bounds are automatically computed (as opposed to user-specified) and relational (as opposed to scalar-valued) bounds, largely based on the Kodkod inclusion described at the end of Section 2.2.

To clarify what we do mean by TACO tight bounds we will now provide some further insight on how TACO models Java class fields in Alloy and Kodkod. We adopt the modeling of Java fields originally introduced in [17].

Let us consider the simplistic Java classes for binary trees presented in Fig. 3. (We will only show how to model class fields; translation of code is explained in [12].) From the classes shown in Fig. 3, TACO produces the Alloy signatures presented in Fig. 4. According to Alloy semantics,

```
public class BinTree {
   TreeNode root; }
public class TreeNode {
   TreeNode left;
   int key;
   TreeNode right; }
```

Figure 3: Sample Java classes representing binary trees.

```
abstract sig Object {}
one sig null {}
sig BinTree extends Object {
    root : TreeNode + null }
sig TreeNode extends Object {
    left  : TreeNode + null,
    key   : Int,
    right : TreeNode + null }
```

Figure 4: Alloy signature hierarchy for binary trees.

the `abstract` signature `Object` only contains objects from its extending signatures. Signature fields, such as `root` or `left` are modeled as total functions (more precisely, due to the relational nature of Alloy, as total functional relations). Recalling that `in` denotes set inclusion and `->` denotes Cartesian product, the typing of fields lets us infer that

```
  root in BinTree  -> (TreeNode + null),
  left in TreeNode -> (TreeNode + null),
   key in TreeNode -> Int,
 right in TreeNode -> (TreeNode + null).
```

Note that, as a result, Alloy fields become *bounded* in a relational sense – each field must be fully contained within some concrete set of pairs.

As explained at the end of Section 2.3, if we analyze a BinTree method with a scope of 5 TreeNode, the following signatures are added to the instrumented model:

```
one sig TN0, TN1, TN2, TN3, TN4 extends TreeNode {}
```

Using the trivial upper bounds derived from typing as a starting point, TACO automatically computes *tight* bounds by removing pairs that could never be part of a field due to violations of the class invariants. The implementation of the bound-tightening procedure is explained in [12, 13].

In terms of our running example, several spurious pairs can be identified as such, and thus automatically removed from the upper bound by the TACO tool. For instance, since the class invariant forces binary trees to be acyclic, pairs like TN0->TN0, ..., TN4->TN4 can never be present in left or right, and are therefore deleted.

Similarly, the symmetry-breaking predicates impose an ordering in the way TreeNode objects relate to each other. In our example, the symmetry-breaking axioms force all nodes to be labeled as per a breadth-first search traversal, with TN0 as the root node. Hence, the pair TN0->TN2 can never be contained in field left: if any node is to be the left child of TN0, it must be TN1. Similarly, all pairs TN0->TNi (for $i > 1$) are automatically removed as well.

For red-black trees (a variant of binary trees) with scope 5 in signature TreeNode, TACO reduces the total number of pairs bounding fields root, left and right from 66 to 22.

Bound tightening heavily profits from the constraints introduced by the class invariant: the richer the class invariant, the more effective the technique will be. We consider this a strength rather than a limitation of the technique: the savings tend to favor more intricate code.

In essence, what we mean by *TACO tight bound* is a set of pairs that bounds the possible interpretations of signature fields and does not contain any spurious pair.

Our interest in tightening field bounds is because during translation to a propositional formula each feasible pair is mapped to a fresh propositional variable. The tighter the bounds, the fewer propositional variables will need to be allocated to represent the resulting formula. Since the worst-case time complexity of the SAT problem is exponential in the number of variables, this reduction is beneficial [12].

3. MUCHO-TACO

MUCHO-TACO is the novel technique that we introduce in this article. It is a simple yet very effective technique for parallelizing bug-finding. It profits from the existence of TACO tight bounds, and allows us to split a given problem into a number of independent subproblems.

We describe the technique in Section 3.1. In Sections 3.2–3.4 we present optimizations that allow for significant reductions in the number of generated subproblems. In Section 3.5 we discuss some of the most relevant implementation details. Finally, in Section 3.6 we prove the soundness and completeness of the technique.

3.1 The MUCHO-TACO Technique

The hypothesis behind the TACO technique is that

> *the fewer propositional variables required to represent the initial state of a system, the better the performance of the SAT-solver during analysis.*

This conjecture has already been tested in the context of TACO, and the results were reported in [12]. The MUCHO-TACO technique takes another step in the same direction by attempting to further reduce the number of propositional variables present in the initial state.

In Section 2.4 we described TACO tight bounds. As a running example, we show in Fig. 5 the bounds for fields left and right for red-black trees with up to 5 Node.

left in	right in
N0->N1 + N0->null + N1->N3 + N1->null + N2->N3 + N2->N4 + N2->null + N3->null + N4->null	N0->N1 + N0->N2 + N0->null + N1->N3 + N1->N4 + N1->null + N2->N3 + N2->N4 + N2->null + N3->null + N4->null

Figure 5: TACO tight bound for fields left and right from red-black trees. Scope 5.

Let us concentrate on field left; the same reasoning will apply to field right. The field is contained in a binary relation on signature Node which, due to the bound tightening procedure implemented in TACO, is substantially smaller than Node $\times$ (Node + null). This is the upper bound relation $U_{\texttt{left}}$ we will use in Kodkod. Notice that $U_{\texttt{left}}$ is nondeterministic: for instance, node N0 is related both to node N1 and to the value null. At the same time, signature fields modeling Java fields have to be total functions. Thus, each instance considered by the SAT-solver contains total functional instances of field left. Due to the TACO bounds, these functional instances must be contained in $U_{\texttt{left}}$.

The MUCHO-TACO technique splits the original model by reducing the nondeterminism in each field's bounds. In our example, this could mean removing some of the nondeterminism from $U_{\texttt{left}}$ and $U_{\texttt{right}}$. In practice, this is carried out by choosing a number n of nodes (let us choose $n = 2$ for our example) and ensuring the removal of any nondeterministic choices for nodes $\texttt{N}_0$ through $\texttt{N}_{n-1}$. Before further details, let us introduce some notation.

NOTATION 3.1. *Given a node* Ni *and a field* f*, we denote by* $Ran(\texttt{N}i, \texttt{f})$ *the set* $\{x \mid \texttt{N}i \to x \in U_{\texttt{f}}\}$.

Once the number n is chosen, we proceed as follows: For each $0 \le i < n$ we select a single $e_i \in Ran(\texttt{N}i, \texttt{f})$ and remove all pairs with domain Ni, with the sole exception of pair $\texttt{N}i \to e_i$. We thus generate a set of upper bounds, each tighter than $U_{\texttt{f}}$. The tighter upper bounds for field left obtained by removing nondeterminism from nodes N0 and N1 (since we chose $n = 2$) are shown in Fig. 6.

N0->N1 +N1->N3 +N2->N3 +N2->N4 +N2->null +N3->null +N4->null	N0->null +N1->N3 +N2->N3 +N2->N4 +N2->null +N3->null +N4->null	N0->N1 +N1->null +N2->N3 +N2->N4 +N2->null +N3->null +N4->null	N0->null +N1->null +N2->N3 +N2->N4 +N2->null +N3->null +N4->null

Figure 6: Tighter upper bounds for field left.

More generally, an initial bound B can be split into some number of tighter bounds $B_1, \ldots, B_k$. The new bounds produced by this algorithm define state spaces that do not intersect. Let us consider the first two bounds for left shown

in Fig. 6, calling them B_1 and B_2, respectively. Any configuration consistent with B_1 must satisfy N0.left = N1, while configurations consistent with B_2 must satisfy N0.left = null. The resulting k models can be analyzed concurrently.

n	#subprob.
2	36
3	720
4	14,400
5	604,800
6	33,868,800

Figure 7: Number of subproblems generated as n increases. Red-black trees, scope 15.

Since the amount of available hardware will always be limited, only a small number of nodes may be subjected to elimination of nondeterminism, lest the number of generated subproblems render the approach infeasible. For instance, Fig. 7 shows the progression for our red-black trees example; notice how fast the number of subproblems increases. Hence we maintain the number of subproblems that may be generated, which indirectly bounds n, below a certain limit. We discuss how to adjust this limit in Section 3.5.

However, if we remove nondeterminism only a small number of nodes at a time, it could be the case that too many models resulting from a split still remain too hard to be tackled directly (that is, within acceptable impact on total user-perceived runtime). We therefore associate a timeout with each analysis. If an attempt to solve a subproblem SP times out, the associated bound B_{SP} is used to split it again using the MUCHO-TACO technique. We discuss election of timeout values in Section 3.5.

The algorithm is implemented using a master/worker architecture. High-level pseudocode outlining the behavior of both kinds of processes is shown in Fig. 8.

```
MASTER
  split initial problem
  do:
    on task request from a worker:
       mark worker as idle
    on any open branch and any idle worker:
       assign branch to worker
       mark branch as active, worker as busy
    on SAT report from a worker:
       send immediate abort msg to any busy workers
       report SAT verdict and send model to user
    on UNSAT report from a worker:
       mark branch as closed
    on TIMEOUT for an active branch:
       send abort msg to worker
       mark and enqueue branch to be split
    on few branches open and any to be split:
     1. using tasks queue load, determine number of
        input nodes in bound to make deterministic
     2. generate new open subproblems with tighter bounds
        by removing nondeterminism from the chosen nodes.
        If no more nondeterminism remains in the bounds,
        enqueue the subproblem without a timeout.
  until all branches closed or any SAT branch found


WORKER[i]
  do:
    send task request to master
    on new task assignment:
       launch solver on light subproblem
       if result is UNSAT:
         report UNSAT_EASY result to master
       otherwise:
         launch solver on full subproblem
         while not (solver finished or timeout reached):
           wait for solver result or msg from master
         report SAT, UNSAT or TIMEOUT result to master
 until exit-to-shell message received from master
```

Figure 8: Pseudocode for Master and Workers.

3.2 Optimization 1: SBP-Guided Configurations

The number of subproblems generated when splitting TACO bounds using a naive algorithm grows too fast. Fig. 7 shows that our Alloy model for red-black trees, using a scope of 15 Node, would need to be split into more than 30 million subproblems for as little as $n = 6$.

As seen in Section 2.3, the intermediate Alloy model is instrumented with symmetry-breaking predicates. These predicates impose a canonical ordering on the nodes in the models considered by the SAT-solver. For instance, for binary trees, the tree in Fig. 9(a) is valid according to the axioms, but the one depicted in Fig. 9(b) is not. The latter includes several violations, namely, the root node is not N0, some nodes are missing (if a node is reachable, any nodes with smaller indices must be as well), and nodes are not sorted in breadth-first-search order.

We propose to restrict the splitting procedure so as to produce only configurations that satisfy the symmetry-breaking axioms. This modification implies, for instance, that the number of subproblems generated for red-black trees drops from 33,868,800 to 36,136. More importantly, since it requires no user input, it has been fully automated.

Table 1 reports reductions achieved on the number of generated subproblems using scope 20, for some of our benchmark classes. The reductions are significant, often exceeding two orders of magnitude and sometimes even reaching three.

3.3 Optimization 2: Prevent Forbidden Aliasing

Many object-oriented classes contain fields that cannot produce aliases. For instance, fields left and right in binary trees will never reference the same object. Note that such conclusions cannot always be inferred from structural properties of a class; they usually follow from the class invariant. In our benchmark, fields left and right cannot be aliased in classes BSTree, RBTree, and AVLTree. Similarly, fields sibling and child cannot be aliased in class BinHeap, and field next cannot be aliased (with itself) in class SLList.

We propose to mine this kind of information about potential aliasing in an automatic fashion, determining whether aliasing may or may not arise

- when f_1 and f_2 are different fields whose codomains intersect, and
- when f is a field and we consider different inputs for f (in this case we check whether a single field can produce aliases).

The first situation occurs when we consider fields like left and right, or root and left, in tree-like structures. The second situation arises when we consider a single field, such as next in list-like structures, or field left in tree-like ones, but different input values.

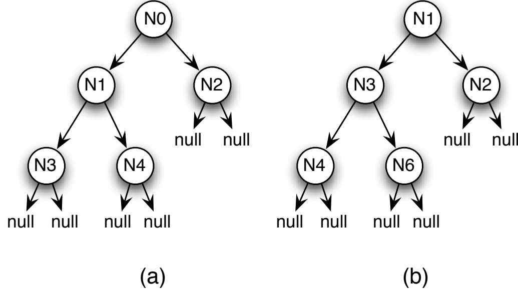

Figure 9: Binary trees satisfying (a) / violating (b) the symmetry-breaking axioms.

To find out whether two different fields $f_1 : \mathtt{T}_1 \to \mathtt{T}$ and $f_2 : \mathtt{T}_2 \to \mathtt{T}$ can yield aliases, we check the following assertion (`FReach`, defined in [12], characterizes the set of objects reachable in a memory heap from its roots):

```
assert differentFields { all a : T1, b : T2 |
   a + b in FReach implies no (a.f1 & b.f2) }
```

To determine whether a field $f : \mathtt{T}' \to \mathtt{T}$ can produce aliases with itself, we check the following assertion:

```
assert singleField { all disj a, b : T' |
   a + b in FReach implies no (a.f & b.f) }
```

In Section 4.3 we will show experimental results estimating the additional time needed to automatically mine such information before starting the main analysis. As an example, our tests for class `BSTree` involved considering the following pairs of fields:

- `root` and `root`, `root` and `left`, `root` and `right`, `left` and `right`, `left` and `left`, and `right` and `right`.

In all cases the answer was negative (no aliasing possible). If we now consider class `TreeSet`, field `parent` forces us to also check whether it would be possible to introduce aliasing through the pairs of fields:

- `parent` and `root`, `parent` and `left`, `parent` and `right`, and `parent` and `parent`.

In all these latter cases we determined that aliasing can be introduced. Since automated aliasing detection should take place before the actual parallel analysis of the original problem can be carried out, we can assume that cluster nodes assigned to the job are still idle, i.e. available for us to run the preliminary aliasing analyses in parallel. This is why in Table 2 we report, for each class and scope, the maximum analysis time required over all different combinations that need to be considered. Since their number is usually small (in our benchmark, no more than 10 for any given class and fixed scope), it is expected that, at least, all the analyses for a given class and scope can be run in parallel.

Once aliasing information is determined, MUCHO-TACO will only generate subproblems whose bound configurations respect those aliasing restrictions. Table 1 shows the impact on the number of subproblems generated after enabling this optimization. As was the case for the optimization from Section 3.2, we use a scope of 20 `Node` objects in all cases. In most of them, further reductions of over one order of magnitude in the number of generated subproblems are achieved.

3.4 Optimization 3: Remove Spurious Subproblems

Let us consider the TACO tight bound for red-black trees with scope 7 shown in Fig. 10(a). In Fig. 10(b) we present a tighter bound obtained by applying the MUCHO-TACO techniques on in, with $n = 5$.

The bound shown in (b) satisfies the conditions enforced by both of the optimizations proposed so far. As shown in Fig. 11, the partial heap on the nodes `N0`, $\ldots$, `N5` respects the ordering imposed by the symmetry-breaking axioms, and no aliasing occurs.

(a)

```
left in N0->N1 + N0->null
 + N1->N3 + N1->null
 + N2->N3 + N2->N4 + N2->N5
   + N2->null
 + N3->N5 + N3->N6 + N3->null
 + N4->N5 + N4->N6 + N4->null
 + N5->N6 + N5->null
 + N6->null

right in N0->N1 + N0->N2
   + N0->null
 + N1->N3 + N1->N4 + N1->null
 + N2->N3 + N2->N4 + N2->N5 +
   + N2->N6 + N2->null
 + N3->N5 + N3->N6 + N3->null
 + N4->N5 + N4->N6 + N4->null
 + N5->N6 + N5->null
 + N6->null
```

(b)

```
left in N0->N1
 + N1->N3
 + N2->null
 + N3->null
 + N4->null
 + N5->N6
   + N5->null
 + N6->null

right in N0->N2
 + N1->null
 + N2->N4
 + N3->N5
 + N4->null
 + N5->N6
   + N5->null
 + N6->null
```

Figure 10: (a) TACO bounds for red-black tree. (b) Tighter bounds generated by MUCHO-TACO using $n = 5$.

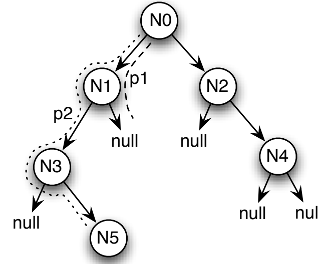

Figure 11: A bound configuration generated using the optimizations from Sections 3.2 and 3.3.

That said, notice that this partial heap instance cannot yield a red-black tree. According to the class invariant, a binary tree is a *red-black* tree if each node is either labeled *red* or *black*, the root node is labeled black, no two consecutive nodes on any path are red, and all paths from the root node contain the same number of black nodes.

Since the path p_1 (`N0` $\to$ `N1` $\to$ `null`) can contain at most two black nodes, the following two cases arise: (1) `N1` is black, and consequently, either `N3` or `N5` on path p_2 must be black. Hence, path p_2 contains at least three black nodes, violating the invariant. (2) `N1` is red, and path p_1 has one black node. Since `N1` is red, `N3` must be black. But then path p_2 contains at least two black nodes, violating the invariant.

Some of the subproblems generated, despite satisfying the symmetry-breaking axioms and abiding by aliasing constraints for their class, result in bounds that systematically violate the class invariant.

This motivates our third optimization, which can be summarized as follows: before analyzing the full version of a subproblem SP, run a quick preliminary test to determine satisfiability of the class invariant, using the MUCHO-TACO bound contained in SP (but excluding the translation of any implementation code).

If no instance is found, we know that further analysis of SP is unnecessary: since no valid inputs exist, no valid inputs can lead to an invalid state.

The proposed preliminary analysis is a substantially easier problem – we have yet to come across any case where obtaining a final (SAT or UNSAT) answer to such a test requires more than a small fraction of a second.

Table 1 shows how the number of subproblems to be generated drops considerably for the most constrained case studies in our benchmark. This is particularly visible in cases like `TreeSet`, `AVLTree` and `BinHeap`.

Table 1: Reductions in the number of subproblems generated as automatic optimizations are enabled. NO = no optimizations. OP1 = optimization 1 only. OP2 = optimizations 1 and 2 combined. OP3 = optimizations 1, 2 and 3 combined.

Class	Opt	n=2	n=3	n=4	n=5	n=6
SLList	NO	4	8	16	32	64
	OP1	3	4	5	6	7
	OP2	3	4	5	6	7
	OP3	3	4	5	6	7
BSTree	NO	72	1,440	43,200	1,814,400	101,606,400
	OP1	26	166	1,258	10,846	103,684
	OP2	13	43	148	526	1,912
	OP3	13	43	148	526	1,912
TreeSet	NO	36	720	14,400	604,800	33,868,800
	OP1	9	56	250	3,028	36,163
	OP2	7	19	52	184	694
	OP3	7	19	34	100	172
AVLTree	NO	36	720	8,640	259,200	14,515,200
	OP1	9	56	105	716	11,670
	OP2	7	19	34	94	334
	OP3	7	19	28	70	142
BinHeap	NO	72	648	10,368	259,200	6,480,000
	OP1	26	108	600	4,204	17,030
	OP2	13	36	111	370	1,034
	OP3	7	11	13	15	17

3.5 Implementation Details

In this section we describe some details that had significant impact on the engineering of MUCHO-TACO.

3.5.1 Translation Overhead Issues

Translating each newly generated subproblem from the Alloy intermediate representation to clausal form is undesirable, due to the considerable overhead involved and the fact that most of it can actually be avoided. Nevertheless, at least during early design and prototyping, letting each worker invoke its own instance of the Alloy Analyzer tool as needed may seem simplest. The load should thus be reasonably distributed. Besides, estimating its total impact should be a simple matter. As it turns out, it is not. This approach does not scale well on multi-core-PC clusters, where full utilization of available hardware requires multiple instances of Alloy to (potentially) be run at any time within the same machine. This may result in occasional interactions between unrelated activity peaks, often injecting enough noise to visibly pollute experimental results. Over long runs, amplified noise tends to snowball to such an extent that it becomes virtually indistinguishable from signal.

3.5.2 Impact on Workflow

The aforementioned issues are easily mitigated but hard to eliminate. Furthermore, the fact that merely launching a JVM/Alloy/Kodkod/JNI/Solver toolchain is several orders of magnitude more resource-demanding than spawning a Minisat process is unlikely to change. This eventually drove us to adopt a lower-level workflow. MUCHO-TACO now uses the Alloy toolchain during initialization only. The master process invokes Alloy to translate the original problem and class invariant to CNF, and all processes henceforth operate directly on formulas at the clausal level. Once in place, this reduced the per-solving overhead from seconds or even minutes to just milliseconds, while load spikes, spurious interactions and user-perceptible noise disappeared.

3.5.3 Dynamic Fanout Control

As mentioned in Section 3, we enforce a limit on the number of subproblems to be generated during the bound-tightening process. This in turn imposes a bound on the number of nodes from which nondeterminism is removed in the TACO bound. MUCHO-TACO uses two queues: a *waitingQ* for tasks waiting for a worker, and a *toSplitQ* holding problems that, having reached their timeouts, await being split into easier ones. The number of tasks in *waitingQ* is kept within two bounds; a lower bound on the size of the queue determines when problems waiting to be split should actually be split. This balances splitting of problems in *toSplitQ*.

3.5.4 Self-adjusting Timeouts

In Section 3.1, where the MUCHO-TACO technique was introduced, we mentioned that a timeout is used to stop ongoing attempts to close a branch after some amount of computational effort fails to produce a verdict. The actual timeout used in the reported experiments is dynamically set when a problem is added to *waitingQ*. In order to explain how this value is set, let P be the set of subproblems that have already been solved. For each subproblem $p \in P$, we have $TO(p)$ (the timeout assigned to subproblem p when it was added to *waitingQ*), and $AT(p)$ (the actual time that it took to analyze subproblem p, possibly smaller). For a new problem np that is being added to *waitingQ*, the timeout is defined by:

$$TO(np) = 5 * \sum_{p \in P} \frac{TO(p)}{AT(p)} \,. \qquad (1)$$

Initial timeouts are set to 40 seconds. An upper cap to values produced by formula (1) is preset at 240 seconds.

3.6 Soundness and Completeness

Under the assumption that the translation of code implemented by TACO is correct, Theorems 3.1 and 3.2 show that the MUCHO-TACO technique neither misses faults nor

reports false-positives. Proofs are not included due to space limitations.

Theorem 3.1. *(*soundness*) Given a problem produced by MUCHO-TACO for which the solver returns SAT, we may retrieve a trace exhibiting a failure in the code under analysis.*

Theorem 3.2. *(*completeness*) If the code under analysis has a fault that can be detected using TACO, then the fault will be detected using MUCHO-TACO as well.*

4. EVALUATION

This section is organized as follows. In Section 4.1 we describe the container classes that we used to evaluate MUCHO-TACO. In Section 4.2 we describe the computing infrastructure that we used in the experiments. In Section 4.3 we present the experimental data and discuss the results.

4.1 Experimental Case Studies

Our benchmark comprises the following classes: An implementation `SLList` of sequences based on singly-linked lists; `BSTree`: an implementation of binary search trees used as part of a benchmark in [29]; `RBTree`: the implementation of class `TreeSet` from `java.util`, based on red-black trees; `AVLTree`: an implementation of AVL trees obtained from the case study used in [2]; `BinHeap`: an implementation of binomial heaps used as part of a benchmark in [29].

4.2 Hardware and Software Infrastructure

Except where otherwise indicated, experiments were run on the CeCAR [32] cluster, which consists of 56 identical quad-core PCs featuring two Intel Dual Core Xeon 2.67 GHz processors with 2 MB of L2 cache per core and 2 GB main memory per host. Parallel analyses were run as 16x4x10h jobs (16 nodes dedicated for up to 10 hours) using one process per core (1 master + 63 workers). Parallel analyses reported in Table 4 were run as described above except for the number of nodes (8x4x10h, 16x4x10h, 24x4x10h, etc). Sequential analyses, such as those using TACO *after* computing tight bounds, were run on a single dedicated node.

4.3 Experimental Results

All the experiments were run 5 times, and the average time is reported. Table 2 reports the analysis time required for the automatic mining of aliasing information proposed in Section 3.4. For any class and scope in the benchmark, the additional time needed barely contributed to the overall analysis time. The only cases in Table 2 requiring over one minute are those of class `BSTree` for scopes 15 and above. For every such scope of said class, the analysis time using TACO and reported in Table 3 exceeds the 10-hour timeout. Moreover, even the analysis times using MUCHO-TACO reach the timeout or are close to reaching it. For example, consider `BSTree.find()` at scope 15, where MUCHO-TACO requires 495:41 while the corresponding aliasing computations take 01:07.

Table 3 presents experimental results for the methods in each of the container classes described in Section 3.2. For each method and each scope we report three values.

Table 2: Time needed for mining aliasing-related information.

Class	s10	s12	s15	s17	s20
SLList	00:02	00:02	00:03	00:03	00:04
BSTree	00:06	00:17	01:07	02:43	21:07
AVLTree	00:03	00:04	00:08	00:14	01:18
TreeSet	00:04	00:05	00:08	00:12	00:31
BinHeap	00:04	00:05	00:08	00:10	00:14

The top row shows the sequential analysis time as reported by TACO in [12]. The middle row shows the analysis time that we measured using MUCHO-TACO. The bottom row estimates the speed-up obtained by using MUCHO-TACO instead of TACO. All times are expressed in `mm:ss` format, and a timeout, noted TO whenever reached, is set at ten hours. In all cases, loops were unrolled up to 10 times.

Table 3: TACO sequential time, MUCHO-TACO parallel time, and achieved speed-up. Loop unrolls: 10.

Class	Method	s10	s12	s15	s17	s20
SL	contains	00:05	00:06	00:07	00:09	00:16
		00:07	00:11	00:03	00:04	00:06
		0.7x	0.5x	2.3x	2.2x	2.7x
	insert	00:07	00:08	00:13	00:26	00:41
		00:08	00:03	00:02	00:04	00:06
		0.9x	2.6x	6.5x	6.5x	6.9x
	remove	00:11	00:12	00:17	00:33	01:01
		00:06	00:03	00:05	00:12	00:15
		1.8x	4.0x	3.4x	2.7x	4.1x
BST	find	114:47	TO	TO	TO	TO
		01:03	13:36	495:41	TO	TO
		109.3x	>44.1x	⋙1.2x		
	add	TO	TO	TO	TO	TO
		333:57	TO	TO	TO	TO
		≫1.8x				
	remove	32:59	TO	TO	TO	TO
		02:08	33:35	TO	TO	TO
		15.4x	>17.8x			
AVL	findMax	00:03	00:04	00:09	00:13	01:09
		00:23	00:10	00:39	00:32	00:38
		0.1x	0.4x	0.2x	0.4x	1.8x
	find	00:36	01:41	08:20	33:06	179:54
		00:25	00:50	01:29	03:16	19:43
		1.4x	2.0x	5.6x	10.1x	9.1x
	insert	04:47	21:53	173:57	TO	TO
		02:05	05:04	62:45	581:57	TO
		2.3x	4.3x	2.8x	≫1.0x	
TSet	find	01:39	06:17	93:17	260:56	TO
		00:58	00:40	04:31	16:48	516:28
		1.7x	9.4x	20.6x	15.5x	≫1.1x
	insert	TO	TO	TO	TO	TO
		08:37	56:41	241:52	TO	TO
		>69.6x	⋙10.6x	≫≫2.5x		
	remove	196:58	TO	TO	TO	TO
		13:03	92:29	TO		
		15.1x	≫6.5x			
BH	min	00:14	00:17	01:31	02:51	07:26
		00:14	00:36	01:13	01:27	03:15
		1.0x	0.4x	1.2x	2.0x	2.3x
	decrKey	30:26	TO	TO	TO	TO
		04:35	13:48	20:51	81:05	236:42
		6.6x	>43.5x	⋙28.8x	≫≫7.4x	⋙≫≫2.5x
	insert	37:30	218:13	TO	TO	TO
		08:13	19:02	21:05	114:36	325:28
		4.6x	11.4x	>28.4x	⋙5.2x	≫≫1.8x
	extrMin	36:52	TO	43:33	176:47	TO
		07:01	26:19	01:46	04:05	01:50
		5.2x	>22.8x	24.6x	43.3x	⋙1058x

The table shows many cases where MUCHO-TACO succeeds in analyzing code for which TACO fails to produce a final verdict within the ten-hour timeout.

Let us briefly focus on method `TreeSet.insert()`. Since sequential analysis exceeds the ten-hour timeout for scope 10, we know that the speed-up achieved by MUCHO-TACO exceeds 69.6x – yet we do not know by how much. If we now proceed to scope 12, we can see that sequential analysis (predictably) exceeds the ten-hour timeout again. However, since analysis time usually grows exponentially as scope is increased, it is reasonable to infer that it does so much more blatantly. While we can only guarantee a speed-up of 10.6x, an actual speed-up well over 100x is more likely. In cells where the speed-up is most probably under-estimated, we used a chain of '>' symbols to denote that the actual value is (most probably) substantially larger than reported. We use as many '>' as there were previous or equal scopes yielding a TO in TACO. Notice, in particular, the use of $\gg\gg\gg$ at scope 15 for `TreeSet.insert()` (where scopes 10 through 15 yielded a TO). In contrast to the reported 2.5x, the actual speed-up is probably several thousand times. To test this hypothesis, we ran the experiment again on a more powerful workstation with the following characteristics: Intel(R) Core(TM) i5-750 CPU running at 2.67GHz, 8 GB of RAM, and Debian 6 OS. We set a new timeout at 200 hours, which was reached before the analysis with TACO could produce a result. This raises the speed-up that we can safely guarantee to at least 50x.

In [12], TACO found a previously unknown bug in an implementation of method `BinHeap.extractMin()` used in [29] as part of a benchmark for Java Pathfinder. To expose the bug, the input binomial heap must involve at least 13 nodes. The highlighted portion of Table 3 shows the speed-ups obtained by using MUCHO-TACO: 20x in the case where TACO can also find the bug, and $\gg$1058x for scope 20, where TACO cannot find the bug within the ten-hour timeout while MUCHO-TACO succeeds in less than two minutes.

In cases where sequential analysis can be completed very efficiently, such as the `SList` class atop Table 3, MUCHO-TACO does not produce significant speed-ups.

As we explained in Section 4.2, all parallel experiments described so far were run on 16 quad-core PCs using one master and 63 worker processes. In table 4 we report the evolution of elapsed analysis times as the number of workers is increased. We chose `AVLTree.find()` for this experiment because its sequential analysis time using TACO was the largest on Table 3 where scope 20 did not reach the ten-hour timeout.

Table 4: Analysis time and speed-up for growing number of workers. Method `AVLTree.find()`, scope 20.

Hardware	8x4	16x4	24x4	32x4	40x4
Workers	31 W	63 W	95 W	127 W	159 W
AVLTree.find	23:14	19:43	13:07	10:07	9:41
Speed-up	7.7x	9.1x	13.7x	17.78x	18.5x

4.4 Threats to Validity

We evaluated MUCHO-TACO on a benchmark consisting of several container classes. Container classes have become ubiquitous [23, 29], and are representative of a wider class of programs that include, for instance, parse trees and XML documents. Moreover, a number of analysis tools have used these classes as benchmarks as well [23, 25, 5, 10, 17, 29]. Also, they are good examples of code in which strong heap properties must be enforced. These classes consist of just a few hundred lines of code each, e.g. ~450 LOC for TreeSet, ~270 LOC for BinHeap, etc. Yet the difficulty of analyzing such classes does not reside in their size, but rather in the complexity of their invariants.

We have strived to realistically estimate the benefits of MUCHO-TACO over the sequential TACO tool, but covering all dimensions with equal emphasis is not possible. One might contest our interpretation of experimental results, for instance, by pointing out that its emphasis on speed-up and elapsed (wallclock) time disregards other important issues like efficiency and total cumulated CPU usage, in terms of which the same payoffs may look less impressive in some cases. While this objection is a sensible one, we believe that a typical potential user of a distributed BEA tool (having, by definition, access to some quantity of idle hardware) will pragmatically favor significant boosts in turnaround time and maximum tractable problem size.

5. RELATED WORK

A small number of code analysis tools are available for experimental comparison. In [12] we compared TACO with JForge [10], Kiasan [9], Java PathFinder [28], ESC/Java2 [6], Jahob [4] and Dafny [19]. Since MUCHO-TACO outperforms TACO, it transitively outperforms these tools.

Even fewer tools are available allowing the parallel analysis of Java programs. In [25], a parallel version of Symbolic Java PathFinder was presented, which reports

> ...*"a maximum analysis time speedup of 90x observed using 128 workers, and a maximum test generation speedup of 70x using 64 workers."*

In Table 3 we showed maximum speed-ups of 1058x for a faulty method and 109.3x, for a correct one, using 63 workers. These figures, as we explained in Section 4.3, are conservative estimations. Several of the case studies from [25] are the same ones used in this article. Furthermore, while [25] addresses the problem of test generation, we face the more demanding one of bounded analysis, where the state space to be traversed is generally larger.

Since TACO operates by reduction to the propositional satisfiability problem, an alternative to MUCHO-TACO would be the use of a distributed SAT-solver as a back-end to TACO. Our attempts to explore this design were hindered by the scarcity of real-world implementations.

CryptoMiniSat2 is an award-winning open source solver with sequential and parallel operation modes. The author also mentions distributed solving among its long-term goals. No public release or other news about this have been announced. GrADSAT [7] reported experiments showing an average 3.27x and a maximum 19.9x speed-up using various numbers of workers ranging between 1 and 34. C-sat [20] is a SAT-solver for clusters. It reports linear speed-ups, but the tool is not available for experimentation.

PMSat [14], an MPI-based, cluster-oriented SAT-solver is indeed available for experimentation, but reports generally small speed-ups. It is based on a notion essentially equivalent to that of *guiding paths*, a concept originally introduced by PSATO [30]. SAT-solvers based on this approach

split the state space by choosing truth values for a number of propositional variables. As explained in [30], if k variables are chosen, 2^k subproblems may be produced. This is not the case in MUCHO-TACO. If inside a MUCHO-TACO bound for a field f we have `f in Ni->M1 + Ni->M2 + ... + Ni->Mk` eliminating nondeterminism for node $\mathtt{N}_i$ requires splitting the bound into k smaller bounds. In the process, the k variables corresponding to the pairs get their value set to either true or false in each model. If we are eliminating the nondeterminism from two distinct nodes $\mathtt{N}_i$ and $\mathtt{N}_j$ in the domain at once, this produces at most as many problems as the size of the Cartesian product $Ran(\mathtt{N}i, \mathtt{f}) \times Ran(\mathtt{N}j, \mathtt{f})$, which is polynomial on the number of propositional variables whose truth value is being set. The procedure can be generalized to an arbitrary number of nodes. This is possible because fields are functional: in other words, we exploit information from the problem domain to improve the analysis. MUCHO-TACO thus generates fewer subproblems than SAT-solvers based on guiding paths.

If instead of the TACO translation we used one that, like the one presented in [27], requires a logarithmic number of variables to represent the codomain of a Java field, then the number of subproblems generated by a parallel SAT-solver based on guiding-paths would be comparable to the number of subproblems generated by MUCHO-TACO. Unfortunately, such a translation does not allow to refine bounds and is incompatible with TACO and MUCHO-TACO.

JForge [10] is very close to TACO in its intentions. In [12] we experimentally compared TACO with JForge, and showed that the TACO technique produces a significant speed-up. Since MUCHO-TACO outperforms TACO, it improves over JForge as well. JForge is available for download, and its authors have been very supportive.

Miniatur [3], like MUCHO-TACO, performs scope-bounded analysis. Some of the examples reported are also treated in this article with MUCHO-TACO. For instance, [3] also analyzes the `TreeSet` implementation, but does so using scope 4 for `Object`, 3 loop unrolls, and a simplified property where only the size is checked to evolve correctly. A distinguishing feature of Miniatur is its improved (with respect to the Alloy Analyzer) handling of integers and arrays. Quoting [3],

> *"This allows us to represent integers in much larger ranges than previously possible. Our tool can handle 16-24 bit integers, a considerable improvement over previous approaches which handled 4 bits."*

MUCHO-TACO can handle 32-bit integers, 64-bit longs and IEEE-754-compliant floating point, thanks to its adoption of the encoding used in FAJITA, our tool for test generation [1]. Since the case studies do not involve arithmetic, we used Alloy integers. The translation from code to a relational logic specification presented in [3] (which slices the translated code according to the specification) could also be used in the translation from MUCHO-TACO. Miniatur is not available for experimentation, even for academic purposes.

Parallel analysis of code has also been attempted by splitting the program control flow graph and using JForge to analyze each slice [22]. The experiments presented intersect with ours. For example, singly linked lists and binary search trees are considered, but for generally small scopes and fewer loop unrolls. For method `add` in class `SLList`, the scope is set to 8 (our benchmark starts with a minimum of 10), and the maximum number of loop unrolls is 8, while we use 10. Similar situations arise in the rest of the examples; notwithstanding, analysis times with the parallel version are substantially larger than those achieved using MUCHO-TACO. For instance, insertion on a binary search tree with scope 7 and 3 loop unrolls when the original problem is split into 4 subproblems, requires 76 seconds for the subproblem that is solved more efficiently and 6409 seconds for the most demanding subproblem [22]. Analyzing the whole problem using MUCHO-TACO requires 23 seconds. The technique presented in [22] is compatible with MUCHO-TACO.

An approach to parallelizing scope-bounded analysis based on data-flow analysis was presented in [21]. The technique is compatible with MUCHO-TACO. The authors also include collection implementations among their experiments. Method `contains` from an implementation of singly linked lists is analyzed, using a scope of 8 nodes and up to 8 loop unrolls. Analyzing the same method with MUCHO-TACO using the same scope and number of loop unrolls takes under one second. Regarding method `insert` from class `RBT` (red-black trees), and quoting the authors,

> *For the RBT.insert() method, VARDEF strategy completes within 40 minutes for a scope of 4 and 4 unrolls while all other strategies run out of memory.*

This method is analyzed by MUCHO-TACO in under one second. MUCHO-TACO can handle said method for scopes up to 15 and 10 loop unrolls without running out of memory.

6. CONCLUSIONS AND FURTHER WORK

We presented a novel technique for parallel analysis of sequential Java code. MUCHO-TACO, the analysis tool implementing the technique, effectively improves code analyzability by allowing TACO users with access to (even modest-sized) PC clusters to harness aggregated computational resources, enabling verification of most problem instances of the kinds that TACO was designed to address at scopes hitherto considered untractable. We now look forward to evaluating MUCHO-TACO well beyond the TACO baseline, on a broader benchmark including many other real-world Java classes. In particular, due to the dramatic speed-ups typically obtained when a bug does exist, we are looking forward to carrying out further experiments with faulty code. Its success at finding bugs also suggests that MUCHO-TACO could be useful as a component within an efficient tool for parallel generation of test inputs.

Some additional optimizations should be considered. For example, it is possible to remove even more propositional variables by using data-flow analysis to prune not just variables from the initial state, but also those from intermediate states of the program under analysis [8].

7. ACKNOWLEDGEMENTS

This publication was made possible by grant NPRP-4-1109-1-174 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

8. REFERENCES

[1] Abad P., Aguirre N., Bengolea V., Ciolek D., Frias M.F., Galeotti J., Maibaum T., Moscato M., Rosner

N., Vissani I., *Tight Bounds + Incremental SAT = Better Test Generation under Rich Contracts*, to appear in Proceedings of ICST 2013.

[2] Belt, J., Robby and Deng X., *Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-based Analyses*, FSE 2009, pp. 355–364.

[3] Dolby J., Vaziri M., Tip F., *Finding Bugs Efficiently with a SAT Solver*, in ESEC/FSE'07, pp. 195–204, ACM Press, 2007.

[4] Bouillaguet Ch., Kuncak V., Wies T., Zee K., Rinard M.C., *Using First-Order Theorem Provers in the Jahob Data Structure Verification System.* VMCAI 2007, pp. 74–88.

[5] Chandrasekhar Boyapati, Sarfraz Khurshid, Darko Marinov: *Korat: automated testing based on Java predicates.* ISSTA 2002: 123-133.

[6] Chalin P., Kiniry J.R., Leavens G.T., Poll E. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2.* FMCO 2005: 342-363.

[7] Wahid Chrabakh and Rich Wolski, *GrADSAT: A Parallel SAT Solver for the Grid*, in UCSB Computer Science Technical Report Number 2003-05.

[8] Cuervo-Parrino B., Galeotti J.P., Garbervetsky D., Frias M.F., *A dataflow analysis to improve SAT-based program verification*, to appear in Proceedings of SEFM 2011.

[9] Deng, X., Robby, Hatcliff, J., *Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs*, in SEFM 2007, pp. 273-282.

[10] Dennis, G., Chang, F., Jackson, D., *Modular Verification of Code with SAT.* in ISSTA'06, pp. 109–120, 2006.

[11] Flanagan, C., Leino, R., Lillibridge, M., Nelson, G., Saxe, J., Stata, R., *Extended static checking for Java*, In PLDI 2002, pp. 234–245.

[12] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, Marcelo F. Frias: *Analysis of invariants for efficient bounded verification.* ISSTA 2010: 25–36.

[13] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, Marcelo F. Frias: *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds*, submitted to IEEE TSE, 2013.

[14] Luís Gil, Paulo Flores and Luís Miguel Silveira: *PMSat: a parallel version of MiniSAT*, Journal on Satisfiability, Boolean Modeling and Computation 6 (2008) 71-98.

[15] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. *Alcoa: the alloy constraint analyzer.* In Proceedings of International Conference on Software Engineering, Limerick, Ireland, 2000.

[16] Jackson, D., *Software Abstractions.* MIT Press, 2006.

[17] Jackson, D., Vaziri, M., *Finding bugs with a constraint solver*, in ISSTA'00, pp. 14-25, 2000.

[18] Khurshid S. and Marinov D., *TestEra: Specification-Based Testing of Java Programs Using SAT.*, Automated Software Engineering 11(4): 403–434 (2004)

[19] Leino K.R.M., *Specification and verification of Object-Oriented Software*, Lecture Notes from Marktoberdorf International Summer School 2008.

[20] Kei Ohmura and Kazunori Ueda, *c-sat: A Parallel SAT Solver for Clusters*, in SAT 2009, LNCS 5585, 2009.

[21] Danhua Shao, Divya Gopinath, Sarfraz Khurshid, Dewayne E. Perry, *Optimizing Incremental Scope-Bounded Checking with Data-Flow Analysis.* ISSRE 2010: 408–417.

[22] Danhua Shao, Sarfraz Khurshid, Dewayne E. Perry: An Incremental Approach to Scope-Bounded Checking Using a Lightweight Formal Method. FM 2009: 757–772.

[23] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, Darko Marinov: Testing Container Classes: Random or Systematic? FASE 2011: 262-277.

[24] Siddiqui J.H., and Khurshid S., *PKorat: Parallel Generation of Structurally Complex Test Inputs*, in Proceedings of ICST'09.

[25] Matt Staats, Corina S. Pasareanu: *Parallel symbolic execution for structural test generation.* ISSTA 2010: 183-194

[26] Torlak E., Jackson, D., *Kodkod: A Relational Model Finder.* in TACAS '07, LNCS 4425, pp. 632–647.

[27] Vaziri, M., Jackson, D., *Checking Properties of Heap-Manipulating Procedures with a Constraint Solver*, in TACAS 2003, pp. 505-520.

[28] Visser W., Havelund K., Brat G., Park S. and Lerda F., *Model Checking Programs*, ASE Journal, Vol.10, N.2, 2003.

[29] Visser W., Păsăreanu C. S., Pelánek R., *Test Input Generation for Java Containers using State Matching*, in ISSTA 2006, pp. 37–48, 2006.

[30] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. 1996. PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. 21, 4-6 (June 1996).

[31] `http://www.mfrias.com.ar/`

[32] `http://cecar.fcen.uba.ar/`